Learning logic programs with negation as failure

Learning Logic Programs with Negation as Failure

F. Bergadano
University of Messina,
Salita Sperone, S. Agata, Messina, Italy,
bergadan@di.unito.it
tel: (+39) 90 393229, fax: (+39) 90 393502

D. Gunetti
University of Torino,
corso Svizzera 185, 10149 Torino, Italy,
gunetti@di.unito.it
tel: (+39) 11 7429216, fax: (+39) 11 751603

M. Nicosia, G. Ruffo
University of Catania,
via A. Doria 6/A, 95100 Catania, Italy,
{nicosia,ruffo}@dipmat.unict.it
tel: (+39) 95 330533, fax: (+39) 95 330094

Abstract. Normal logic programs are usually shorter and easier to write and understand than definite logic programs. As a consequence, it is worth investigating their learnability, if Inductive Logic Programming is to be proposed as an alternative tool for software development and Software Engineering at large. In this paper we present an extension of the ILP system TRACY, called TRACY^{not}, able to learn normal logic programs. The method is proved to be sound, in the sense that it outputs a program which is complete and consistent w.r.t. the examples, and complete, in the sense that it does find a solution when it exists. Compared to learning systems based on extensionality, TRACY and TRACY^{not} are less dependent on the kind and number of training examples, which is due to the intensional evaluation of the hypotheses and, for TRACY^{not}, to the possibility to have restricted hypothesis spaces through the use of negation.

1 Introduction

It is well known that definite logic programs have the computational power of Turing machines [2]. However, experienced Prolog programmers normally use much more than just the simple syntax of definite clauses. In particular, clauses containing negative literals in the body are widely used, resulting in the so-called *normal logic programs* [16]. As we are interested in the construction of program development and Software Engineering tools based on Machine learning techniques [6] (and see also the papers by Idestam-Almquist [13], Jorge and Brazdil [14], and Mofizur and Numao [18] in this

book), we believe the ability to learn normal programs to be an important step towards that achievement.

Normal logic programs are widely used just because they are much easier to devise, write and analyze, than definite programs. Many theoretical considerations could be done on this issue, but we prefer to take a practical stand: normal logic programs are shorter and clearer than definite programs, because negative knowledge can be expressed through what is already known. Consider, for example, the *intersection* program, returning the intersection of the two input lists X and Y. Such a program must check whether an item occurring in X also occurs in Y, or not. To this end, two subprograms *member* and *notmember* are needed. If negation is allowed, we have just to devise a program for *member*, and then set: "notmember(A,B) :- not member(A,B)." If negation is not allowed, the two subprograms must be treated as independent concepts, and a program for *notmember* must be developed, too. Since negation can make programs sensibly shorter, this may have a positive influence on their learnability, as the difficulty of learning a given logic program is very much related to its length.

As usual, we want a learned logic program to behave correctly, at least on the given examples E^+ and E^-. Moreover, we want the learning system to be able to output only correct programs and to be able to find them whenever they exist in the hypothesis space HS. These requirements are formally grasped by the following definitions:

Definition 1 *A program P is* complete *w.r.t. E, iff* $\forall e^+ \in E\ P \vdash e^+$.

Definition 2 *A program P is* consistent *w.r.t. E, iff* $\forall e^- \in E\ P \nvdash e^-$.

Definition 3 *An induction procedure M is* sound *iff whenever M terminates successfully and outputs a program P belonging to HS, then P is complete and consistent w.r.t. E.*

Definition 4 *An induction procedure M is* complete *iff whenever a complete and consistent program w.r.t. E exists in HS, then M will output one such program.*

In the ILP literature, different approaches to handling negation can be found. [23] is a beautiful theoretical extension of inverse resolution [20] to normal clauses, but the approach turns out to have many practical drawbacks. Moreover, we feel bottom-up methods (such as also an improved version of CLINT [9], which is able to handle negation) not particularly suited for program assistants and Software Engineering applications [6]. An approach based on a three-valued logic can be found in this book, in [17]. This approach overcomes the main limitations of extensional methods (discussed below), but it has been applied only to single predicate learning, whereas multiple predicate learning is essential for Software development. Also, extensive tests on medium size/complexity normal programs are still missing.

Extensional top-down systems too, such as FOIL [21] and its derivatives, are able to learn clauses containing negative literals (i.e., normal clauses). However, this is different from learning normal programs. In fact, to achieve efficiency, extensional systems generate candidate program clauses one at a time and check them against the examples independently of one another. For instance, the clause "p(X) :- q(X,Y), not p(Y)." is normally said to cover the example p(a) if there is a positive example q(a,b) of q such that p(b) is a negative example of p (or it is not known to be a positive example). Other clauses for p and q (e.g. those learned previously) are not used to try

a derivation for q(a,b) and p(b). In other words, clauses are evaluated extensionally at the time of learning.

However, since clauses are learned extensionally, but then the whole program is interpreted intensionally (i.e. it is run on a Prolog interpreter), extensional methods present some major drawbacks. First, they can fail to find a complete and consistent program w.r.t. the given examples, even if it exists in the hypothesis space. Second, they can output a program which is inconsistent w.r.t. the given examples. In other words they are not sound nor complete. These problems can be avoided only if some special examples are given to the system, or if the system is able to query the user for the missing examples, as in the FILP system [4]. Nonetheless, some (or even many) of the given examples may be unnecessary, in the sense that even a subset of them would be sufficient to learn the desired program. As a consequence, the learning system may waste a lot of time by trying to cover examples which are, in fact, useless. A thorough discussion of these problems can be found in [4]. Moreover, the kinds of programs that are learned are usually very simple and often limited to clauses defining just one predicate. Few systems [22, 15, 19, 10, 4, 11] are able to learn programs for multiple predicates, while even simple Prolog programs contain clauses with different predicates in the head. Again this is due, in part, to the fact that clauses are learned one at a time and independently of each other. If we want to learn a program for predicates P and Q, and we try to construct a clause antecedent for P where Q occurs, then Q must have been defined by the user, or determined extensionally, by means of all of its relevant examples.

The above problems are particularly serious in a Software Engineering setting. Especially in this case, the potential ILP user should be able to get a logic program each time it exists in the designed hypothesis space (and not only sometimes or even often), and the synthesized program should be correct (and not only "approximately" correct) at least on the seen examples. Moreover, the user should not be compelled to provide an extensionally complete list of examples, since otherwise it could be simpler and/or faster to write down directly the required program.

In this paper we present a system, called TRACY^{not}, able to fulfill the above requirements by adopting an intensional evaluation of candidate clauses, and able to learn normal logic programs. In TRACY^{not}, clauses are checked against positive and negative examples by running them with a Prolog Interpreter. In this way the problems of extensionality are automatically overcome, since a (normal) logic program is learned in the same way as it will be used. The rest of the paper is organized as follows. In sections 2 and 3 we briefly review the TRACY system, thoroughly described in [5], in order to present the adopted approach. In section 4, the shortcomings of the basic approach when learning normal programs are discussed, and section 5 contains the description of TRACY^{not}, able to learn normal logic programs. TRACY^{not} is proved to be complete and sound in section 6. Section 7 contains an example of the use of TRACY^{not} with a normal logic program of medium complexity. In section 8 we discuss our approach and conclude.

2 Review of the Basic TRACY Algorithm

Searching a space of possible programs is clearly unfeasible. As a consequence, TRACY only consider partial programs made up of clauses successfully used to prove at least one positive example (we will also call such partial programs *traces*). Partial programs

are put together and checked against the negative examples. If some negative example is derived, backtracking occurs. The learning algorithm is automatically suitable for multiple predicate learning and it can work with any number of positive and negative examples. In particular, in no sense it requires an extensionally complete set of positive examples.

The complexity of the method is not as good as for extensional approaches, that only need to enumerate allowed clauses, and evaluate them independently. However, it is not as bad as enumerating all allowed programs (sets of clauses), because only the sets of clauses that are derivation traces for some positive example need to be examined. If the examples are sufficiently simple and well-chosen, this means a dramatic improvement.

TRACY is not the first intensional approach to the ILP problem (see, e.g., [22, 20, 10, 12]). However, two distinct features of our TRACY are that (a) the induction procedure does not require any interaction with the user and does not rely on any special provided example in order to be sound and complete, and (b) it can learn sets of clauses instead of just one clause at a time.

To describe TRACY, we need the following definition:

Definition 5 *Given a set of clauses S and an example e such that $S \vdash_{SLD} e$, a clause of S is* successful *(w.r.t. e) if it used in the proof of e.*

TRACY works as follows: The actual set of candidate clauses (CC) is produced from a description of the hypothesis space (HS), and is given in input to TRACY. For each positive example e^+, TRACY looks for a set of successful clauses (a partial program) deriving that example. The partial program p is added to the partial programs discovered previously, and the covered examples are removed, until no more positive examples remain. At every step, the whole set of clauses learned up to that point is checked against the negative examples, and if some of them are derived the learning task backtracks to a different derivation for e^+. Here is an informal description of TRACY:

TRACY:

input: a set of positive examples E+
a set of negative examples E-
a description of the hypothesis space HS
a background knowledge BK
CC $\leftarrow$ generate_all_clauses(HS)
P $\leftarrow \emptyset$
while E+ $\neq \emptyset$ do
for each e^+ such that CC $\cup$ BK $\vdash_{SLD} e^+$ do
let $p \subseteq$ CC be the set of clauses successfully used
in the derivation of e^+
P $\leftarrow$ P $\cup$ p
if $\exists$ e^- such that P $\cup$ BK $\vdash_{SLD} e^-$ then backtrack
E+ $\leftarrow$ E+ - e^+

We note that candidate recursive clauses must satisfy some well-order relation, to guarantee the termination of learned programs, and for making the learning procedure terminate. That is, we require the set of clauses in CC form a terminating program, since otherwise the test CC $\cup$ BK $\vdash_{SLD} e^+$ could not terminate. As a consequence, all

candidate recursive clauses must satisfy the following pattern:

R(Y1, ..., Yi, ..., Yn) :- ..., *wor*(Z,Yi), ..., R(Y1, ..., Z, ..., Yn), ...

where *wor* is any well-order relation, which makes every recursive call of the clause one step closer to the termination. As an example, *cons*(A,B,C) determines the well ordering |B| < |C|, while *tail*(A,B) determines the well ordering |B| < |A|. Obviously even more sophisticated techniques can be adopted, which apply to sets of clauses and not just to a single clause. A thorough discussion of such techniques can be found in [7].

3 Example

We show in details the behavior of TRACY on a very simple example: learning *member*. Let CC containing the following three clauses:

CC =
{
c_1 = member(X,Y) :- tail(Y,Z), null(Z).
c_2 = member(X,Y) :- head(Y,H), tail(Y,T), member(X,T).
c_3 = member(X,Y) :- head(Y,X). }

The background knowledge BK will contain the usual definitions for all of the predicates above, except obviously for *member*. The examples used are the following:

e_1^+ = member(a,[a]), e_2^+ = member(a,[c,b,a]),
e_1^- = member(a,[]), e_2^- = member(a,[b,c]).

1) CC $\vdash e_1^+$ (we omit the background knowledge for brevity, and we use $\vdash$ for $\vdash_{SLD}$), by using c_1. As a consequence, we set P = $\{c_1\}$, and since P does not derive any of the negative examples we can continue. Since P $\nvdash e_2^+$, the learning process goes on.

2) Now, CC $\vdash e_2^+$ and the first successful clauses used to derive e_2^+ are c_1 and c_2, so we set P = $\{c_1, c_2\}$. However, P $\vdash e_2^-$. Hence, we backtrack to a different partial program for e_2^+, and we find $\{c_2, c_3\}$. However, P = $\{c_1, c_2, c_3\}$ is still inconsistent (P $\vdash e_2^-$), and since backtracking to a different partial program for e_2^+ fails (there is not such a program in CC), we must backtrack to the first positive example. P is empty again.

3) We discover that e_1^+ can be derived from c_3, and we set P = $\{c_3\}$, which is consistent with the negative examples. Then, e_2^+ is derived from c_1 and c_2, and we set P = $\{c_1, c_2, c_3\}$. Now P $\vdash e_2^-$, and backtracking to a different partial program for e_2^+ allows to discover the partial program $\{c_2, c_3\}$. Now, P = $\{c_3\} \cup \{c_2, c_3\} = \{c_2, c_3\}$. At this point, P derives all the positive examples and no negative examples, and represents a legal program for *member*.

We observe from the above example, that an extensional system would have been unable to learn *member* with the examples provided, since member(a,[c,b,a]) is not extensionally covered by the recursive clause of *member*. In fact, the positive example

member(a,[b,a]) would be required. Note also that *member* could have been learned by using only the second positive example (i.e. the one involving recursion).

Clearly, we could also define a hypothesis space containing clauses with different consequents. As a consequence, multiple predicate learning is automatically achieved.

4 Trying to Learn Normal Programs with TRACY

The soundness and completeness of TRACY (proved in [5]) are based on monotone reasoning, which is in force when dealing with pure definite clauses. But, what happens when negation is brought into consideration? In fact, derivation is no longer monotone, and TRACY can fail to work correctly. Two main problems come out, that will be discussed in the next subsections. As an illustrating example, we will use the problem of learning *intersection*, together with its main subprogram *member*. As a hypothesis space (i.e., a set of candidate clauses), we will use the set of clauses reported in figure 1.

```
1) int(X,Y,Z):-null(X),null(Z).
2) int(X,Y,Z):-null(Z).
3) int(X,Y,Z):-null(Z),head(X,X1),member(X1,Y).
4) int(X,Y,Z):-null(Z),head(X,X1),not member(X1,Y).
5) int(X,Y,Z):-head(X,X1),tail(X,X2),member(X1,Y),int(X2,Y,Z).
6) int(X,Y,Z):-head(X,X1),tail(X,X2),member(X1,Y),int(X2,Y,W),assign(W,Z).
7) int(X,Y,Z):-head(X,X1),tail(X,X2),member(X1,Y),
                int(X2,Y,W),cons(X1,W,Z).
8) int(X,Y,Z):-head(X,X1),tail(X,X2),not member(X1,Y),int(X2,Y,Z).
9) int(X,Y,Z):-head(X,X1),tail(X,X2),not member(X1,Y),
                int(X2,Y,W),assign(W,Z).
10) int(X,Y,Z):-head(X,X1),tail(X,X2),not member(X1,Y),
                int(X2,Y,W),cons(X1,W,Z).
11) int(X,Y,Z):-head(X,X1),tail(X,X2),int(X2,Y,W),cons(X1,W,Z).
12) int(X,Y,Z):-tail(X,X2),int(X2,Y,Z).
13) int(X,Y,Z):-tail(X,X2),int(X2,Y,W),assign(W,Z).
14) member(X,Y):-head(Y,X).
15) member(X,Y):-head(Y,X),tail(Y,Tail),member(X,Tail).
16) member(X,Y):-head(Y,KW),tail(Y,Tail),member(KW,Tail).
17) member(X,Y):-tail(Y,Tail),member(X,Tail).
```

Figure 1: A hypothesis space for learning *intersection*

The background knowledge will contain the usual definitions for *null*, *head*, *tail*, *cons*, and *assign*[1], while negation is defined as usual in Prolog:

```
not(X) :- X,!,fail.
not(X).
```

Finally, the following positive and negative examples of *intersection* will be used[2]:

[1]For example, *assign* is defined as: assign(X,X).

[2]Note that we do not provide any positive or negative example for *member*.

e_1^+: int([a],[b,a],[a])
e_2^+: int([b,a],[a],[a])
e_1^-: int([a],[a],[])
e_2^-: int([],[a],[a])
e_3^-: int([a],[b,a],[])
e_4^-: int([b],[a],[b])

4.1 Useful but not successful clauses

If TRACY is given in input the above information, it will output the trace made up of clauses 1, 7, 8, 14, 17, that represents a legal solution for *intersection* and *member*. However, consider what happens if the positive examples are given in input (and hence taken into consideration by TRACY) in reverse order: The learning process starts by searching for a derivation for e_2^+. Example e_2^+ can be derived from the hypothesis space using only clause 14 for *member*, plus some clauses for *intersection*. Clause 17 is not necessary, and is not learned. However, clause 17 is required in order to avoid the derivation of the negative examples.[3] As a consequence, no consistent traces are found for example e_2^+, and the learning task fails. In fact, TRACY is unable to find a program that is consistent w.r.t. example e_2^+ and the negative examples. Here, the problem is that derivation is no longer monotone, and it can be the case that a negative example is derived by a set of clauses (a trace) but not by a superset of those clauses, as above. But TRACY is only able to learn programs that are traces, i.e., where every clause in the program is used to derive at least one positive example. It is unable to learn programs where some clauses are used only to avoid undesired derivations.

In this case, the above problem could be remedied by testing different example orderings. But, in the worst case, $n!$ different orderings of a set of n positive examples must be checked, which could be practically unfeasible. Moreover, in general, the above solution won't work. Suppose the only given positive example is e_2^+. Again, clauses 1, 7, 8, 14, and 17 represent a correct solution program that cannot be found because clause 17 cannot be learned from e_2^+. Of course, in this case we have no different example orderings to test for.

4.2 "Always true" Antecedents

Suppose we add to the set of clauses of figure 1 the clause:

18) member(X,Y):-true.

Then, TRACY will never be able to learn clause 8. The problem, again, lies in the non-monotonicity of SLDNF derivation. Clause 18 is always successful and, as a consequence, negative literals for *member*, as those in the body of clause 8, will always fail. No clause containing a negative literal for *member* can be successfully used to derive any positive example, and hence such a clause cannot be learned.

The above situation, due to clause 18, is somewhat artificial, since clauses with an empty body can be easily avoided when designing a hypothesis space. However, the problem remains. It could be in general difficult, or even impossible, to determine if a

[3] For example, the first candidate trace discovered by TRACY is the set of clauses S = {1, 7, 8, 14}. It is easy to check that S $\vdash e_3^-$. However, S $\cup$ {clause 17} $\nvdash e_3^-$.

hypothesis space contains clause antecedents that are *always evaluated to true* w.r.t. the given examples and background knowledge. If such a clause "$P(X_1, ..., X_n) : -body$" is in the hypothesis space, then no clause of the form "$head : -\alpha, notP(T_1, ..., T_n), \beta$" can be learned by TRACY.

5 TRACY^{not}

The problems illustrated in the previous section are due to the fact that TRACY is based on the properties of SLD derivation, which is monotonic: if e can be derived by a set of clauses S, it will be derived by any superset of S. But this is no longer true when negative literals are introduced and SLDNF derivation is activated, which is nonmonotonic. To solve this problem, we propose the following solution: instead of modifying TRACY in order to be able to deal with nonmonotonic reasoning (in this case, negation as failure), we go back to SLD derivation, but handle negative literals encountered in the bodies of candidate clauses in a special way. This is formalized in what we call a TRACY^{not} derivation, which is defined as follows:

Definition 6 *Let S be a set of clauses and e and example. We say that $S \vdash_{TRACY^{not}} e$, with the sequence of goals $e = G_0$, G_1, ..., $G_n = \square$, if the current goal G_{i+1} is obtained from goal G_i in one of the following two ways:*

1. *if $G_i = R, L_2, ..., L_n$, and R is a positive literal, then G_{i+1} is obtained by applying SLD resolution to G_i (i.e., R is resolved with the first matching clause found in S);*
2. *if $G_i = not\ Q, L_2, ..., L_n$, then $G_{i+1} = L_2, ..., L_n$, and the procedure side_effect(Q,σ) is called (see below). σ is the computed answer substitution.*

Hence, a TRACY^{not} derivation differs from the classical SLDNF derivation because of the way negative literals are treated. In TRACY^{not}, a negative literal *not Q* is always immediately successful, but, as a side effect, Q is treated as follows:

side_effect(Q,σ):

- If S $\vdash_{TRACY^{not}} e^+$, then $E^- \leftarrow E^- \cup$ Qσ (i.e., Qσ is a new negative example that must not be covered by the solution program);
- If S $\vdash_{TRACY^{not}} e^-$, then $E^+ \leftarrow E^+ \cup$ Qσ (i.e., Qσ is a new positive example that must be covered by the solution program; no backtracking to a different trace is required in this case. There can be different derivations of e^- from S, and for each of them the corresponding Qσ must be added to E^+);

The effects of side_effect(Q,σ) must be retracted in case of backtracking of the main procedure (i.e., examples added as a consequence of side_effect(Q,σ) must be removed).

The rationale for the above way of handling negation should be clear: if the derivation of a positive example e^+ from a program P requires proving (with negation as failure) "not Q", this is equivalent to require that P $\nvdash$ Q (i.e., Q is a negative example w.r.t. P). Conversely, if P $\vdash e^-$, and a negative literal "not Q" is involved, we can avoid such a derivation by looking for another program P' such that P' $\vdash$ Q (i.e., Q is a positive example w.r.t. P'). By assuming the input set of candidate clauses to be a

terminating program (see section 2), we are sure that a finite number of positive and negative examples will be added, and that the learning process will terminate (either successfully or not).

The shown approach also overcomes the problem of "always true" antecedents. A negative literal is immediately evaluated to true, and hence "always true" antecedents cannot influence the learning process (in fact, "always true" antecedents could be learned because they can be used to derive every positive example, but are later rejected because they also derive every negative example).

In the following section, some formal properties of TRACY^{not} will be proved. We conclude this section illustrating the behavior of TRACY^{not} on the problem of learning *intersection*.

Let be given the same set of candidate clauses CC and the same background knowledge as in section 4, together with the following positive and negative examples of *intersection*:

e_1^+ int([b,a],[a],[a])
e_1^- int([a],[b,a],[])
e_2^-int([a,b],[c,a],[])

As we saw in subsection 4.1, example int([b,a],[a],[a]) does not allow to learn the recursive clause for *member*, which is however required in order to avoid the derivation of the negative examples. Let us show how TRACY^{not} works on this set of examples.

1. The *trace* T_1={1,7,8,14}, is learned from example e_1^+, using the negative literal "not member(b,[a])" from clause 8. "not member(b,[a])" is immediately evaluated to true, but, **side effect 1:** e_3^- = member(b,[a]) becomes a new negative example.

2. T_1 is checked against the negative examples:

   (a) $T_1 \nvdash e_3^-$;

   (b) $T_1 \vdash e_1^-$ using "not member(a,[b,a])".
   side effect 2: e_2^+ = member(a,[b,a]) becomes a new positive example;

   (c) $T_1 \vdash e_2^-$ through literals "not member(a,[c,a])" and "not member(b,[c,a])".
   side effect 3: e_3^+ = member(b,[c,a]) becomes a new positive example[4].

3. No trace can be learned from e_3^+ (i.e., CC $\nvdash e_3^+$), and backtracking to point 2.(c) occurs. e_3^+ is removed from the set of positive examples, and e_4^+ = member(a,[c,a]) is added.

4. From example e_4^+ the trace T_2 = {1,7,8,14,17} is learned. $T_1 \cup T_2 = T_2$ is found consistent with the negative examples; since T_2 also derives e_2^+, no new derivation is attempted and the learning task terminates with a learned program for *intersection* and *member* which is consistent with the given (and added) positive and negative examples.

[4]In general, it is not possible to avoid a negative example to be classified as a positive one, or vice versa, during the learning task. We will come back to this problem in the last section.

6 Soundness and Completeness of TRACY^{not}

To prove the soundness of TRACY^{not} we first need two lemmas relating SLDNF derivation with TRACY^{not}*derivations*[5], as defined by definition 6.

Lemma 1 *if* $T \vdash e$ *then* $T \vdash_{tn} e$.

Proof: If T $\vdash e$ does not involve negative literals, then obviously T $\vdash_{tn} e$, for the definition of $\vdash_{tn}$. If T $\vdash e$ involves negative literals, then T $\vdash_{tn} e$, since negative literals are always successful in $\vdash_{tn}$.

Lemma 2 *If* $T \vdash_{tn} e$ *then* $T \vdash e$, *if* T *derives each of the negative literals involved in* $T \vdash_{tn} e$ *(i.e., TRACY*^{not} *does not derive any of the examples added because of the derivation* $T \vdash_{tn} e$*).*

Proof: If T $\vdash_{tn} e$ does not contain negative literals, then T $\vdash e$, since we are using SLD-derivation. Otherwise, suppose T $\vdash_{tn} e$ involves negative literals "not Q_1", ... "not Q_n". Then, by the hypothesis, T $\vdash$ not Q_i, for $1 \leq i \leq n$, and hence T $\vdash e$.

Now, if we suppose the candidate clauses given in input to TRACY^{not} form a terminating program, we have the following:

Theorem 1 *TRACY*^{not} *is sound.*

Proof: Let TRACY^{not}(CC,E^+,E^-) = P. The learning procedure terminates successfully only if, for none of the negative examples e^-, it is the case that P $\vdash_{tn} e^-$. Hence, for the contrapositive of lemma 1, it turns out that P is consistent. Now, P = $\cup P_i$, such that, $\forall e_i^+ \in E^+$, $P_i \vdash_{tn} e_i^+$. But $\vdash_{tn}$ is a monotonic derivation, and P does not derive any of the negative examples added during the learning task. Hence, by virtue of lemma 2, P $\vdash e_i^+$ for each $e_i^+ \in E^+$, i.e., P is complete.

Theorem 2 *TRACY*^{not} *is complete.*

Proof: Let P be a complete and consistent program in the hypothesis space, w.r.t. the given positive and negative examples E^+ and E^-. Obviously, $\forall$ $e_i \in E^+$, P $\vdash e_i$. $\forall i$, let $P_i \subseteq$ P be such that $P_i \vdash e_i$ and all of the clauses in P_i are successfully used in the derivation. Also, let $P_i' = P_1 \cup P_2 \cup ... \cup P_i$ be consistent for each i. By lemma 1, if $P_i \vdash e_i$ then $P_i \vdash_{tn} e_i$. Hence, TRACY^{not} will sooner or later find such a trace P_i and will add it to P_{i-1}', until $P_i' =$ P.

It remains to show that P_i' is consistent, for each i. We need the following definition:

Definition 7 *Let G be a goal, and let S be a set of clauses such that* $S \vdash_{SLDNF} G$. *Let* $\{Q\}$ = {*not* q_1, ..., *not* q_n} *be a set of negative literals involved in the derivation of G. Since SLDNF derivation is in force, it is the case that* $P \nvdash q_i$ *for* $1 \leq i \leq n$. *We say that* $\{Q\}$ *is* essential *(for the derivation of G) if and only if* $P \cup \{Q\} \nvdash G$ *but* $P \cup q_i \vdash G$ *for* $1 \leq i \leq n$.

Lemma 3 P_i' *is consistent.*

[5] In the following, $\vdash$ will stand for $\vdash_{SLDNF}$, and $\vdash_{tn}$ will stand for $\vdash_{TRACY^{not}}$.

Proof: Suppose $P_i^{'}$ is inconsistent. As a consequence, there must exist and example e^- such that $P_i^{'} \vdash e^-$. There are two cases to consider:

- if $P_i^{'} \vdash e^-$ does not involve negative literals, then we have a contradiction, since $P_i^{'} \subseteq P$, and $P \nvdash e^-$ by hypothesis.
- If $P_i^{'} \vdash e^-$ involves negative literals, let $\{R_1\}$, ..., $\{R_m\}$ be the essential sets occurring in the derivation.
  TRACY^{not} rejects $P_i^{'}$ iff, for each $\{R_i\}$, $\nexists$ $T \subseteq CC$ such that $T \vdash_{tn} \{R_i\}$, and $T \cup P_i^{'}$ is consistent.
  Since P is consistent, $P \nvdash e^-$. Since $P_i^{'} \subseteq P$ and the sets $\{R_1\}$, ..., $\{R_m\}$ are essential for the derivation of e^-, it must be the case that $P \vdash \{R_i\}$ for some i. Hence, by lemma 1, $P \vdash_{tn} \{R_i\}$. But then, $\exists$ $P \subseteq CC$ such that P is consistent and $P \vdash_{tn} \{R_i\}$ for some i. As a consequence, TRACY^{not} cannot derive e^-, against the initial hypothesis.

7 A More Complex Case

In this section we report the performance of TRACY^{not} to learn a normal program whose size is slightly larger than classical ILP test cases. The program receives as input a directed graph and return as output an hamiltonian cycle for the graph, if it exists. This program is described as problem number 31 in [8]. It is worth noting as the program can be kept short and fairly simple because of an extensive use of not.

The hypothesis space used by TRACY^{not} is given through the notation of Clause Sets (see [6] for a description of Clause Sets) and is reported in figure 2 (f(X,Y) means that there is an arc connecting X and Y). Figure 3 reports the background knowledge and the set of forbidden conjunctions of literals.[6] Clauses containing one of these conjunctions will never be true, and hence there is no need to produce them into the actual set of candidate clauses CC given in input to TRACY^{not}.

The hypothesis space of figure 2 describes a space of $2^8 + 2^5 + 2^3 = 296$ clauses. By avoiding the generation of the clauses containing any forbidden conjunctions of literals, and by requiring that each output variable in the head of candidate clauses must be instantiated in the body[7], the actual set of candidate clauses CC, reported in figure 4, is produced.

TRACY^{not} receives in input the set of candidate clauses CC and background knowledge, and the following positive and negative examples:

e_1^+ = hamilton([f(a,b),f(b,c),f(a,c)],[a,c,b])
e_1^- = hamilton([f(a,b),f(b,c)],[a,b])

the first program output by TRACY^{not} is found after around seven seconds (precisely, 7.690 seconds), and is made up of clauses {3,23,31,35}. This program, although not completely correct, is complete and consistent w.r.t. the given examples, and can be used for sometime. The following examples have been added by TRACY^{not} during the

[6] *Forbidden conjunctions of literals* is one of the many constraints that can be enabled in the Clause Sets language to limit the size of the hypothesis space.

[7] To activate this restriction, an input-output mode for each involved predicate must be provided. We omit this information for brevity. More on this issue can be found in [6].

```
{hamilton(G,C):-{ edge({f(U,V),f(U,W)},G),
                  path(U,{V,W},G,C),
                  not path(U,{V,W},G,C),
                  not uncovered(C,G),
                  uncovered(C,G)
                  }.
uncovered(C,G):-{ node({V,U},G),
                  not member(V,C),
                  member({V,U},C)
                  }.

member(A,B):-{ head(B,A),
               tail(B,T),
               member(A,T)
               }.}
```

Figure 2: The hypothesis space for learning *hamilton*.

learning task:

e_2^+ = member(a,[a,c,b])
e_3^+ = member(b,[a,c,b])
e_3^+ = member(c,[a,c,b])
e_2^- = member(a,[b])
e_3^- = member(c,[])
e_4^- = member(c,[a,b])
e_5^- = uncovered([a,c,b],[f(a,b),f(b,c),f(a,c)])

The program equivalent to the one described in [8] can be obtained by using the backtracking facility of the Prolog interpreter[8]. TRACY^{not} finds such a program, composed of clauses {3,27,31,35}, in 32 seconds, adding the same examples as above. Four more examples are added and then retracted during the whole learning task:

member(a,[]) /* classified as positive by TRACY^{not} */
uncovered([a,b],[f(a,b),f(b,c)]) /* classified as positive */
member(a,[a,b]) /* classified as negative */
member(b,[a,b]) /* classified as negative */

8 Discussion and Conclusion

We conclude this paper by discussing two main issues.

Complexity of the induction procedure. In [5], we proved that the computational complexity of TRACY is proportional to $|CC|^{nd}$, where |CC| is the number of candidate

[8] I.e., by asking TRACY^{not} to look for alternative derivations for the input set of positive examples.

```
/******** forbidden conjunctions of literals ********/

!(head(_,_),head(_,_)).
!(not member(_,_),member(_,_)).
!(not uncovered(_,_),uncovered(_,_)).
!(path(_,_,_,_),not path(_,_,_,_)).
!(path(_,_,_,_),path(_,_,_,_)).
!(not path(_,_,_,_),not path(_,_,_,_)).
!(edge(_,_),edge(_,_)).


/******** background knowledge ********/

rel([_|T],[]):-!,fail.
rel([],_):-!.
rel([_|T1],[_|T2]):-rel(T1,T2).

path(U,V,G,P):-path1(U,[V],G,P).

path1(U,[U|P],G,[U|P]).

path1(U,[W|P2],G,P):-rel([W|P2],G),
                      edge(f(V,W),G),not member(V,P2),
                      path1(U,[V,W|P2],G,P).
edge(f(V,W),GR) :- head(GR,X),eq(X,f(V,W)).
edge(f(W,V),GR) :- head(GR,X),eq(X,f(V,W)).
edge(f(V,W),GR) :- tail(GR,G),edge(f(V,W),G).

eq(X,X).
head([H|_],H).
tail([_|T],T).

node(V,G):-edge(f(V,W),G).
```

Figure 3: Forbidden conjunctions of literals and background knowledge for learning *hamilton*

1) hamilton(G,C) :- edge(f(U,V),G).
2) hamilton(G,C) :- edge(f(U,V),G),path(U,V,G,C).
3) hamilton(G,C) :- edge(f(U,V),G),path(U,V,G,C),not uncovered(C,G).
4) hamilton(G,C) :- edge(f(U,V),G),path(U,V,G,C),uncovered(C,G).
5) hamilton(G,C) :- edge(f(U,V),G),not path(U,V,G,C).
6) hamilton(G,C) :- edge(f(U,V),G),not path(U,V,G,C),not uncovered(C,G).
7) hamilton(G,C) :- edge(f(U,V),G),not path(U,V,G,C),uncovered(C,G).
8) hamilton(G,C) :- edge(f(U,V),G),not uncovered(C,G).
9) hamilton(G,C) :- edge(f(U,V),G),uncovered(C,G).
10) hamilton(G,C) :- edge(f(U,W),G).
11) hamilton(G,C) :- edge(f(U,W),G),path(U,W,G,C).
12) hamilton(G,C) :- edge(f(U,W),G),path(U,W,G,C),not uncovered(C,G).
13) hamilton(G,C) :- edge(f(U,W),G),path(U,W,G,C),uncovered(C,G).
14) hamilton(G,C) :- edge(f(U,W),G),not path(U,W,G,C).
15) hamilton(G,C) :- edge(f(U,W),G),not path(U,W,G,C),not uncovered(C,G).
16) hamilton(G,C) :- edge(f(U,W),G),not path(U,W,G,C),uncovered(C,G).
17) hamilton(G,C) :- edge(f(U,W),G),not uncovered(C,G).
18) hamilton(G,C) :- edge(f(U,W),G),uncovered(C,G).
19) hamilton(G,C) :- not uncovered(C,G).
20) hamilton(G,C) :- uncovered(C,G).

21) uncovered(C,G) :- node(V,G).
22) uncovered(C,G) :- node(V,G),node(U,G).
23) uncovered(C,G) :- node(V,G),node(U,G),not member(V,C).
24) uncovered(C,G) :- node(V,G),node(U,G),member(V,C).
25) uncovered(C,G) :- node(V,G),node(U,G),member(V,C),member(U,C).
26) uncovered(C,G) :- node(V,G),node(U,G),member(U,C).
27) uncovered(C,G) :- node(V,G),not member(V,C).
28) uncovered(C,G) :- node(V,G),member(V,C).
29) uncovered(C,G) :- node(U,G).
30) uncovered(C,G) :- node(U,G),member(U,C).

31) member(A,B) :- head(B,A).
32) member(A,B) :- head(B,A),tail(B,T).
33) member(A,B) :- head(B,A),tail(B,T),member(A,T).
34) member(A,B) :- tail(B,T).
35) member(A,B) :- tail(B,T),member(A,T).

Figure 4: Candidate clauses for learning *hamilton*

clauses given in input to TRACY, n is the number of positive examples, and d is the maximum "depth" on all the examples[9]. This complexity stands between an exhaustive search in the space of possible programs (that would be proportional to the powerset of CC, and hence exponential in CC), and a search in the space of possible clauses (linear in the size of CC), typical of the extensional systems. Since TRACY^{not} differs from TRACY only for the treatment of negative literals, it is easy to see that its complexity turns out to be proportional to $|\text{CC}|^{(n+m)d}$, where m is the number of positive examples added by the induction procedure until a solution program is found. Although the complexity of TRACY is exponential in the number of positive examples, most of the time a complete program can be learned using only one well chosen example[10]. Moreover, the examples can be chosen to be as simple as possible, as long as they carry the same information, in order to limit their depth (i.e., parameter d)[11]. See [5] for a thorough discussion of this issue.

In the case of TRACY^{not}, the situation seems to get worse because of the examples added during the learning task. However, this may not be the case for two reasons: (1) if the number and depth of the initial positive and negative examples are small, then the number of added examples is small as well. In fact, an example is added because of the derivation of another example involving a negative literal, and the lower the depth of an example, the lower the number of clauses (possibly containing negative literals) involved in its derivation; (2) the use of negation allows to write shorter and simpler programs and, in general, the shorter a logic program, the easier to learn it. Hence, being able to learn normal logic programs allows to design smaller hypotheses spaces; i.e., the size of CC is likely to decrease when learning a normal logic program instead of an equivalent definite program. Consider, for example, the case study of section 7. If negation is not allowed, we must define two separate concepts for *notuncovered* and *notmember*[12], resulting into a larger hypothesis space and number of candidate clauses CC.

Misclassified examples. During the learning task it may happen that a positive example of the target concept is classified as negative, or vice versa. For example, this is the case for the learning task of *intersection*, in section 5. There, at step 2.(c), member(b,[c,a]) is added as a positive example. In the subsequent steps no derivation is found for such an example, and backtracking occurs. However, in general, it is possible for TRACY^{not} to find a derivation for a negative example misclassified as positive, and to output a solution program containing a trace to derive that example. Nontheless, the learned program still will be correct (i.e., complete and consistent) w.r.t. the positive and negative examples provided initially. Moreover, the more meaningful are the initial examples and the more restricted is the hypothesis space, and the lower are the chances for the above situation.

Automatically misclassified examples can be avoided if we are willing to accept the

[9]Roughly speaking, the "depth" of an example is related to its complexity, e.g., is proportional to the size of the sets or lists that it contains. See [5] for the details.

[10]For instance, this is the case for the program *intersection* in section 5, which is learned by using just one positive example; and for the program *member* in section 3. Obviously the learned program can be different from the one the user "has in mind".

[11]For instance, in section 3, *member* can be learned by using just the second positive example $e_2^+ =$ member(a,[c,b,a]), but a better choice would be, e.g., member(a,[c,a]), whose depth is 2, whereas the depth of e_2^+ is 3.

[12]A possible, very simple, clause set for *notmember* could be, e.g., {notmember(A,B) :- {head(B,H), different(A,H), tail(B,T), notmember(A,T)}, resulting in $2^4 = 16$ new clauses.

introduction of "yes/no" queries to the user. In this case, when a derivation for a negative example is found involving a negative literal "not Q", Q is not immediately added as positive example, but the user is queried to know whether Q *is* a positive example, or not. If the answer is "no", Q is added to the negative examples, and not to the positive. In a similar way we do when a trace for a positive example involving negative literals is found. This procedure can be made automatic if a database of positive and negative examples of the target concept(s) is available, in place of a user.

The possibility of enhancing TRACY^{not} with queries becomes particularly interesting when using the system to generate test cases for a program to be tested [3, 6]. In that case, the program itself is used to classify the added examples during the production of an adequate test set distinguishing the program from all of the alternatives in the hypothesis space.

TRACY^{not} turns out to be a natural evolution of TRACY. Unlike in classical top-down and bottom-up methods, in our approach candidate clauses are not evaluated extensionally, examples are not required to be complete in any sense, queries are not necessary (although they can be useful in TRACY^{not}), and set of clauses are learned at a time. For a comparison of our approach with other systems adopting a pure intensional evaluation of clauses, the reader can refer to [5], whereas a similar way of handling additional positive and negative examples, in the framework of theory revision, can be found in [1].

In this paper, our primary goal was to present a system able to learn correct normal logic programs while keeping the main qualities of the basic learning procedure. The learnability of Prolog programs (as opposed to definite logic programs) goes through the ability to deal with non monotonic reasoning, above all negation as failure. Prolog programmers make an extensive use of negation, and practical tools for software engineering and program assistants must be able to deal with it.

Acknowledgement: This work was in part supported by BRA ESPRIT project 6020 on Inductive Logic Programming.

References

[1] I. adé, L. De Raedt, and M. Bruynooghe. Theory Revision. In S. Muggleton, editor, *Proc. Third Int. Workshop on Inductive Logic Programming*, Ljubljana, Slovenia, 1993. Jozef Stefan Institute.

[2] H. Andreka and I. Nemeti. The Generalized Completeness of Horn Predicate Logic as a Programming Language. *Acta Cybernetica*, 4:3–10, 1978.

[3] F. Bergadano. Test Case Generation by Means of Learning Techniques. In *Proc. ACM SIGSOFT*, Los Angeles, 1993. ACM.

[4] F. Bergadano and D. Gunetti. An Interactive System to Learn Functional Logic Programs. In R. Bajcsy, editor, *Proc. 13th Int. Joint Conf. on Artificial Intelligence*, pages 1044–1049, Chambéry, France, 1993. IJCAII.

[5] F. Bergadano and D. Gunetti. Learning clauses by tracing derivations. In *Proc. 4th Int. Workshop on Inductive Logic Programming*, Bonn, Germany, 1994.

[6] F. Bergadano and D. Gunetti. *Inductive Logic Programming: from Machine Learning to Software Engineering*. MIT Press, Cambridge, MA, 1995.

[7] R. M. Cameron-Jones and J. R. Quinlan. Avoiding Pitfalls When Learning Recursive Theories. In R. Bajcsy, editor, *Proc. 13th Int. Joint Conf. on Artificial Intelligence*, pages 1050–1055, Chambéry, France, 1993. IJCAII.

[8] H. Coelho and J. C. Cotta. *Prolog by Example: How to Learn Teach and Use it.* Springer-Verlag, Berlin, 1988.

[9] L. DeRaedt. *Interactive Concept Learning.* Ph.D. thesis, Katholieke Univ. Leuven, 1991.

[10] L. DeRaedt and M. Bruynooghe. CLINT: A Multistrategy Interactive Concept-Learner and Theory Revision System. In R. S. Michalski and G. Tecuci, editors, *Proc. Workshop on Multistrategy Learning*, pages 175–190, Harpers Ferry, VA, 1991.

[11] L. DeRaedt and M. Bruynooghe. A theory of clausal discovery. In R. Bajcsy, editor, *Proc. 13th Int. Joint. Conf. on Artificial Intelligence*, pages 1058–1063, Chambéry, France, 1993. IJCAII.

[12] L. DeRaedt, N. Lavrač, and S. Džeroski. Multiple Predicate Learning. In R. Bajcsy, editor, *Proc. 13th Int. Joint Conf. on Artificial Intelligence*, pages 1037–1042, Chambéry, France, 1993. IJCAII.

[13] P. Idestam-Almquist. Efficient Induction of Recursive Definitions by Structural Analysis of Saturations. In L. De Raedt, editor, *Advances in Inductive Logic Programming.* IOS Press, 1995.

[14] A. Jorge and P. Brazdil. Architecture for Iterative Learning of Recursive Definitions. In L. De Raedt, editor, *Advances in Inductive Logic Programming.* IOS Press, 1995.

[15] J. U. Kietz and S. Wrobel. Controlling the Complexity of Learning in Logic Through Syntactic and Task-Oriented Models. In S. Muggleton, editor, *Inductive Logic Programming*, London, 1991. Academic Press.

[16] J. Lloyd. *Foundations of Logic Programming.* Springer Verlag, Berlin, 1984.

[17] L. Martin and C. Vrain. A Three-valued Framework for the Induction of General Programs. In L. De Raedt, editor, *Advances in Inductive Logic Programming.* IOS Press, 1995.

[18] C. R. Mofizur and M. Numao. Top-down Induction of Recursive Programs from Small Number of Sparse Examples. In L. De Raedt, editor, *Advances in Inductive Logic Programming.* IOS Press, 1995.

[19] K. Morik. Balanced Cooperative Modeling. In R. S. Michalski and G. Tecuci, editors, *Proc. Workshop on Multistrategy Learning*, pages 65–80, Harpers Ferry, VA, 1991.

[20] S. Muggleton and W. Buntine. Machine Invention of First Order Predicates by Inverting Resolution. In *Proc. of the Fifth Int. Conf. on Machine Learning*, pages 339–352, Ann Arbor, MI, 1988. Morgan Kaufmann.

[21] J. R. Quinlan and R. M. Cameron-Jones. Foil: A midterm report. In P. Brazdil, editor, *Proc. European Conference on Machine Learning*, pages 3–20, Berlin, 1993. Springer-Verlag, LNAI 667.

[22] E. Y. Shapiro. *Algorithmic Program Debugging.* MIT Press, Cambridge, MA, 1983.

[23] K. Taylor. Inverse Resolution of Normal Clauses. In S. Muggleton, editor, *Proc. Third Int. Workshop on Inductive Logic Programming*, Ljubljana, Slovenia, 1993. Jozef Stefan Institute.